TUTOR1.DO
TUTOR1.ASM 01 Nov 84

An Assembly Language Tutorial for the Model 100

By Mike Berro [75765,473]

0. Preface.

I have an ulterior motive for writing this. I want to sharpen my technical writing skills. Any comments or criticisms about contents or style would be appreciated. Also, I will be happy to try to answer any questions you have about assembly language.

1. Switching Gears from Basic to Assembly Language.

After programming in Basic, using assembly language may at first seem like switching from the Model 100 to the HP-41 calculator. Gone are string variables and the print statements. You don't even have a multiply or a divide command like the HP-41. You have to multiply or divide by using successive additions or subtractions. However, the benefits over Basic include high speed and small size.

My game Planet Protector (PLANET.SRC on CompuServe's Model 100 SIG) is only 2.5K long, and the spaceships move reasonably smoothly across the screen. Compare tht to my earlier Basic game Starfighter (STARF.100), which is over 10K long and slower than anything. And doing what is essentially bit-mapped graphics to produce your own shapes would slow Basic down by at least another factor of ten.

Anyway, since you're still reading this you probably already know the advantages of assembly language, so we'll begin.

2. How to Use This Tutorial.

It would be to your benefit to have at least a reference book on 8085 assembly language, if not a textbook. I am not trying to write a book here, so there may be some information left out. This tutorial is designed as an adjunct to a text book, and not a substitute.

It will also be useful for you to have an assembler. See HELP.ASM in XA4 for help in choosing one. The sample programs I use will be formatted for Custom Software's assembler, but only because that happens to be the one I have. You should have no difficulty translationg to your own assembler.

Also it is very important to backup all the files on your M100 before you assemble any program. Basic (almost) always makes sure that it doesn't clobber other files (unless you open a file for OUTPUT instead of INPUT.) If you assign a value to a variable, the value does get stored in memory, but Basic will never store it in a protected area of memory.

Machine language programs have no compunction about storing values anywhere. If you make a typo so that the program stores values where the file directory should be you're in big trouble. And it happens to the best of 'em. Maybe I'm being paranoid, but I'll say it again:

>>>>>> BACKUP ALL YOUR FILES BEFORE YOU BEGIN ASSEMBLY <<<<<<

Our goal for this series is to be able to write a program that allows

you to design any shape, and then move it across the screen. The shape table will be stored in a text file created by the user. The program will be called SHAPE. It will take several installments of TUTOR before we are finished.

3. What an Assembler Does.

An assembler converts your source file, which is a text file, into a machine language file. You create the source file using the built-in text editor. The source file contains the source code that the assembler converts into a series of numbers that represent the instructions you want carried out (machine language.)

The numbers are stored in memory, and then the memory locations used are identified by assigning a filename to them, with the suffix ".CO". The .CO file is called a machine language file.

It is possible to program directly in machine language by poke-ing the numbers directly into memory, but that gets tedious quickly. Machine language is all numbers, no english commands at all. Assembly language consists of "memnonics" that represent the machine language numbers. You only have to remember the memnonics, which by definition are supposed to be easy to remember. Then the assembler converts the memnonics into machine language for you.

The next section deals with binary and hex numbers, and with some of the inner workings of the 80C85.

4. The 80C85 and Number Systems.

At this point you need to know the binary and hexadecimal number systems. In this section I will try to explain why you need these other number systems. If you are not familiar with them, most books about assembly language start with them. If your book does not, it probably is not a beginner's book.

The CPU in the Model 100 is an 80C85 integrated circuit. (The C stands for CMOS, which is a battery conserving type of electrical device.) As such it only responds to electrical signals, no matter how much you yell at it. It is an 8-bit CPU, which means it can respond to eight signals simultaneously. Each signal is a bit. Now the CPU is also a digital device, so each signal can only be a one or a zero. Music is an analogue signal; you can send Beethoven's fifth symphony through one wire (although it sounds better through two.) A digital signal can only be on or off, one or zero.

Given an 8-bit digital CPU, there is a limit to the number of different messages or commands that we can give it at any one time. In fact there are only 256. 00000000 is one message, 00000001 is another, 00000010 is the "next" one, up to 11111111. These are all binary numbers since each digit (bit) can be one of two values. We normally use the decimal numbers from 0-9 (ten values). Assemblers very often use hexadecimal numbers, which use 16 values.

All three number systems are used in assembly language programming. In fact, the only reason to use decimal numbers is because we are familiar with them. The CPU only recognizes ones and zeroes, eight at a time. There are 256 combinations possible. Hexadecimal is used because of convenience.

Look at Table 1. Notice that the left hexadecimal digit precisely

---

Dec	Binary	Hex	Oct
000	00000000	00	000
001	00000001	01	001
002	00000010	02	002
...	.....	..	...

Ø1Ø	ØØØØ1Ø1Ø	ØA	Ø12
Ø11	ØØØØ1Ø11	ØB	Ø13
Ø12	ØØØØ11ØØ	ØC	Ø14
Ø13	ØØØØ11Ø1	ØD	Ø15
Ø14	ØØØØ1110	ØE	Ø16
Ø15	ØØØØ1111	ØF	Ø17
Ø16	ØØØ1ØØØØ	1Ø	Ø2Ø
Ø17	ØØØ1ØØØ1	11	Ø21
...	.....	..	...

TABLE 1. Number Systems

corresponds the left four digits of the binary number, and the right hexadecimal digit corresponds to the right four digits of the binary number. Binary numbers would be appropriate to use, but they take up more memory in your source file (8 digits). Hexadecimal numbers are the next logical choice.

Octal numbers are sometimes recommended for the 8Ø85. The first two digits of the binary number correspond to the first octal digit, the next three digits correspond to the second octal digit, and the last three binary digits correspond to the last octal digit.

I do not recommend that you learn octal arithmetic, although it couldn't hurt. Octal arithmetic is very useful for converting the source code into the actual machine language, but since that is what our assembler is supposed to be doing for us, there is no real need to master it.

TUTOR2.DO

5. The 8ØC85 Architecture.

This section describes what's available for the programmer to use in the 8Ø85. It is written from the programmer's point of view, and not the engineer's. A more complete analysis of the 8Ø85 will be left for later.

One of the most important functions of the CPU is to manipulate data. In Basic we store the data in variables, both numeric and string. We don't know exactly where in memory the data is stored, but we know that Basic knows, so we don't worry about it. In assembly language we have to worry about it.

We can store data directly inside the CPU. The 8Ø85 has seven 8-bit registers. Each register can store a value from $ØØ to $FF (Ø to 255: a dollar sign prior to a number indicates a hexadecimal value.) The registers are designated by single letters: A, B, C, D, E, H and L. They are not all created equal, and are used for different purposes.

The A register is also called the accumulator. All of the 8-bit arithmetic is done on the value in the accumulator. Addition, subtraction, ANDing, EXCLUSIVE ORing, ORing, rotations and comparisons all involve the A register (don't worry if you don't know what they all mean. You can worry if you don't know what the first two mean though.) For example, the instruction ADI $1Ø adds the value $1Ø to what is already in the A register. The instruction ADD B adds the value of the B register to the value of the A register, and then stores the value back into the A register. The B register is unchanged. Only the A register is affected by any of these operations. Now that we know how to do arithmetic on the A register, how do we get a value into it?

There are five ways to get a value into the A register. The first is to load it with a preset value, called an immediate value, like $Ø7, or $AD. The second is to transfer it from a different register. The third is to get the value from somewhere in memory. The fourth is to get it from a peripheral device, and the final method is to get the value from the serial

data port.

The last two methods involve advanced topics. The serial data port is used solely for the cassette input. A peripheral device is something that communicates with the 8085, something that facilitates input and output (I/O). The keyboard, LCD display, modem, speaker, light-pen and printer (with their associated circuitry) are all I/O devices.

Luckily we don't have to learn the I/O instructions, because there are subroutines already built-in that read data from these devices, or send data to them. Some of these subroutines are listed starting on page 79 of the Model 100 Technical Manual. We will return to them in section 9.

6. The MoVe Immediate (MVI) Instructions.

The MVI instructions move an immediate value into any specified register. The following examples should explain how it works better than mere words:

```
MVI     A,$08   ;A will have the value 8 after this instruction
MVI     C,$ED   ;C will have the value $ED after this instruction
MVI     H,$11   ;H will have the value $11 after this instruction
```

Technically speaking, MVI is an instruction by itself, part of the 8085 "instruction set". The grouping following it is called the operand. For any move instruction, including the move immediate, the operand always consists of a source and a destination. The destination always comes first, then a comma, and then the source. The source for the MVI instruction is always an 8-bit number, a value between $00 and $FF.

The destination is always either a register or a memory location. One of the nice features of the 8085 is the ability to treat a memory location just like it was a register. You have to set up certain conditions before you can do so, and that will be covered in section 8.

The MVI instruction takes up two bytes of memory. A byte is an 8-bit value, so each memory location stores one byte. The first byte is the MVI r, part, where r stands for any register. The second byte is the immediate value, the source.

7. The MOVe (MOV) Instruction.

The MOV instruction moves data from one register to another. The instruction format is MOV dr,sr, where dr is the destination register, and sr is the source register. The content of the source register is the same after this instruction is executed. The destination register will contain the same value as the source register.

The MOV instruction takes only one byte of memory.

8. Moving Data to and from Memory.

So far we have moved immediate numbers into registers, and values from register to register. Now we will see how to retrieve data from memory. In this section we will also see what a source program looks like.

To move data into a memory location, we must first decide what address to use. Each of the 65,536 memory locations has a unique address, a number from $0000 to $FFFF. It takes a 16-bit number count that high, and it can be done on the 8085. We have already seen that the 8085 has seven 8-bit registers. Six of the registers can be "doubled-up" to give you three 16-bit registers. The B and the C register give you the BC register, the D and E registers givve you the DE register, and the H and L registers give you, surprise!, the HL register.

The 16-bit registers are not separate from the 8-bit registers. If you store an 8-bit number in L, and then a 16-bit number in HL, the 8-bit number is lost. To add to the confusion, most assemblers allow you to use only the first letter of a 16-bit register. Then the only way to tell the difference between an 8 or 16-bit register is from context, i.e. from the instruction that precedes it. Here we will use both letters for a 16-bit register.

The HL register is the only register that can be used to specify a memory address. It is called a pointer register because it points to a memory location. The H register contains the high byte of the address, and the L register the low byte (that's why it's not called the FG register.)

High and low bytes may be unfamiliar to you. Consider the decimal number 47. From convention we know that 4 is the number of "tens", and the 7 is the number of "units", high and then low. For the hex number $47, 4 is the number of "sixteens", and 7 is the number of "units". In the same way for the number $4768, $47 is the number of "two-hundred-fifty-sixes", and $68 is the number of "units". We split it this way because the first two digits can fit in the high byte (register) and the second two can fit in the low register. However, as we shall see later, sometimes the low byte comes first, with the high byte following, but the HL register is always high-low.

The instruction that loads an address into the HL register is the LXI instruction. The LXI instruction loads a 16-bit immediate value into the register pair designated in the first half of the operand, the destination. The second half of the operand is the 16-bit immediate value. The format is LXI rp,$nnnn, where rp is any of the three register pairs, and $nnnn is a 2 byte (up to four hex digit) number.

Once HL points to a memory address, we can treat that memory location just like it was a register. We can use:

```
MVI     M,$AB   ;memory location pointed to by HL will equal $AB
MOV     A,M     ;A register will have value of memory location
```

The LXI instruction requires three bytes of memory. The first specifies the instruction and the destination register (all in one byte), and the last two contain the immediate 16-bit value. It is interesting (and confusing) to note that addresses in memory are always stored low byte first. The instruction LXI HL,$1234 assembles into the three consecutive bytes $21, $34, $12. $21 is the LXI H, instruction. The $34 and $12 is the address with the low byte first. You don't have to worry about the order when you are programming because the assembler puts all the addresses in the proper order. You do have to worry about it when debugging your program. Sometimes you have to examine the machine language code to find the problem, and this high-low business can get confusing. To repeat, addresses in the registers are always stored high-low. Addresses in memory are stored low-high.

TUTOR3.DO

9. Subroutines and a Simple Program.

Let us now jump into some actual programming. Reading about instructions is all very well, but the only way to learn is to use them.

Let us suppose that we want to print the seconds from the real time clock onto the LCD screen. We will have our program print the word "Time = ", and then print the seconds.

We don't know how the screen works, but browsing through the technical manual, we find a ROM subroutine that seems like it will do most of the work for us. It is a subroutine that Radio Shack calls "LCD". The manual describes it as "Displays a character on the LCD screen at current cursor position." Sounds good, but what does current cursor position mean? We don't know, so we'll just use the subroutine and see where the numbers get displayed. It is now time to back-up every file in memory. Twice.

There is more information in the subroutine description. It says the entry address is (hex) 4B44, entry conditions: A=character to be displayed, exit conditions: None. The entry address is simple the address of the subroutine. In Basic we gosub to a line number. We dont have line numbers here, but we do have memory addresses.

The entry condition tells us that the A register must contain the character we want printed when we call the subroutine. Of course registers don't contain characters, they contain 8-bit numbers. We need to know what number represents each character. The User's Manual tells us that starting on page 211. It gives us the numbers in decimal, hex and binary. We will use hex to be consistent.

Look at program 1, and then we will discuss it.

```
ØØ1     ;PRTIME
ØØ2     ;Oct 3Ø, 1984
;
ØØ3     ORG     $DACØ
ØØ4     ENT     $DACØ
;
ØØ5     MVI     A,$54   ;character code for "T"
ØØ6     CALL    $4B44   ;print character in A onto screen
ØØ7     MVI     A,$49   ;character code for "I"
ØØ8     CALL    $4B44   ;print character
ØØ9     MVI     A,$4D   ;character code for "M"
Ø1Ø     CALL    $4B44   ;print character
Ø11     MVI     A,$45   ;character code for "E"
Ø12     CALL    $4B44   ;print character
Ø13     MVI     A,$2Ø   ;character code for space
Ø14     CALL    $4B44   ;print character
Ø15     MVI     A,$3D   ;character code for "="
Ø16     CALL    $4B44   ;print character
Ø17     MVI     A,$2Ø   ;character code for space
Ø18     CALL    $4B44   ;print character
;
Ø19     LXI     HL,$F934        ;load address of tens of seconds
Ø2Ø     MOV     A,M             ;move tens of seconds into register A
Ø21     CALL    $4B44           ;print character
Ø22     LXI     HL,$F933        ;load address of unit seconds
Ø23     MOV     A,M             ;move unit seconds into register A
Ø24     CALL    $4B44           ;print character
```

```
;
025     CALL    $12CB           ;wait for keypress
026     JMP     $5797           ;jump to menu
;
027     END
```

PROGRAM 1. Print seconds to screen.

Please note that the line numbers are NOT part of the program. I added them here for clarity, but the source code should not have them. Line numbers are often printed out by the assembler itself, when you direct the assembler to produce a listing.

The first thing you may notice are all the semicolons. They indicate that what follows is a comment. Not all assemblers use the semicolon as the comment indicator, so check your manual.

There are no blank lines either. Every line must have at least one character. However, it makes the program easier to read if different portions of the program are separated by blank lines, so the next best thing is to use a comment line without a comment, just a semicolon.

I've also put a comment after every instruction. Comments are great if you have to come back to a program after not working on it for a while, but they do take up space. I would recommend you use comments liberally at first, and then delete some of them if you need more memory.

The second thing you may notice is the ORG and ENT commands. These commands are not commands for the 8Ø85, but are commands for the assembler. The ORG command is a standard (almost universal) command that tells the assembler where the program is to reside in memory. It is the address of the first instruction (or data) of your program (the ORiGin.) In this case the program will start at $DACØ, which is 56ØØØ in decimal.

The highest available memory address for your programs is 62959. Memory above that is used for the file directory, and other RAM data the M1ØØ needs. Never let your program extend past there, and don't write to an address above 62959 unless you know what you're doing.

The ENT instruction tells the assembler where the program should begin execution. I have never found it neccessary to make the ENT address different from ORG. Just make sure the first part of your program is an instruction and not data.

The final instruction is END, and that is also an assembler directive. It tells the assembler that that's all there is. You can put comments after the END command without using a semicolon.

All the other lines contain 8Ø85 assembly code.

Line five is the move immediate command, and moves the value $54 into register A. $54 is the ASCII value for the letter "T". It is important to note that even though all computers use ASCII code, the ASCII values may be different. The ASCII value of "T" on the Apple is $D4.

Line ØØ6 says it prints the character in A onto the screen. What it actually does is CALL the subroutine at $4B44. The CALL instruction is just like the GOSUB command in Basic. When a return instruction (and there are several types) is encountered, program execution will resume at the instruction immediately following the CALL instruction.

In lines ØØ7 to Ø18, each letter of the word "IME = " is moved into the A register, and then $4B44 is CALLed. The next section prints the seconds from the real-time clock onto the screen.

To do this, we need to know where the seconds are located in memory. Information like this can be found on a "memory map" of the Model 1ØØ. A memory map tells what each address or range of addresses is used for. Most commercial assemblers come with one. By looking in Custom Software's rather extensive one, I discover that the tens of seconds are stored at address $F934, and the unit seconds are at $F933.

So now we want to move a value from a memory location into the A register. To do that we must first load the memory address into the HL register (pointer register.) Line Ø19 does that with the 16-bit load immediate instruction. Line Ø2Ø moves the value from memory pointed to by HL ($F934), into the A register, and line Ø21 CALLs the subroutine at $4B44 print the value in A to the screen.

Lines Ø22 through Ø24 do the same thing with the unit seconds at $F933.

Once the seconds have been displayed, we want the program to stop and wait until we press any key before it returns to the menu. Line Ø24 does that for us.

Line Ø24 calls a subroutine at $12CB. $12CB is the entry address for the subroutine that the M1ØØ technical manual calls CHGET, which "waits and gets character from keyboard. Since we don't care what key is pressed, that is all we need to know. As it happens, the ASCII value of the key we press will be in the A register when execution returns from that subroutine.

Line Ø25 will only be executed after a key has been pressed. At line Ø25 execution JuMPs to address $5797. The jump instruction is like the GOTO instruction in Basic. The jump is executed no matter what. $5797 returns execution to the main menu, it is like the MENU command in Basic.

Remember to leave out the line numbers when you type in this program. Be careful to use either tabs or spaces as required by your assembler. They are usually very finicky, and a space where a tab should be, or vice versa, may give you a strange error message when you assemble it.

When you run the program from the menu, the screen clears, and the message we programmed will be displayed in the upper-left corner. Evidently the "current cursor position" described in the technical manual is line 1 and row 1. What happens is that running any program from the menu automatically clears the screen and sets the cursor to the upper-left cornaer of the screen. If you run the program from Basic with a CALL 56ØØØ command, the message will appear wherever the cursor was.

1Ø. Sneak Preview.

That concludes the first chapter of this tutorial. The next chapter will cover conditional branching. The jump instruction used in the previous section is an unconditional branch, it always occurs. Suppose you want to jump back to the beginning of the program if the key pressed was a "B", and jump to the menu if it was not. Two lines should be inserted between lines Ø24 and Ø25:

```
Ø24a    CPI     $42     ;compare A with the immediate value $42 (= ASCII "B")
Ø24b    JZ      $DACØ   ;if equals, then jump to start of this program
```

As you can see, Conditional branching makes things a lot more interesting.

11. Using Labels.

In section 9 a program was presented that displays "TIME = ", followed by the seconds. In this section a more useful way of displaying messages is discussed.

There are several ways to print characters to the LCD. The sample program in section 9 actually uses the most inefficient method. However, the Radio Shack Technical Manual doesn't tell you that there is a ROM subroutine that prints a whole series of characters (a "string") for you.

The subroutine to do that is located at $5A58. I learned this from the instruction manual for Custom Software's Model 100 Assembler. The assembler even has a nifty little "macro" that does it for you. (A macro is a one word abbreviation for any number of instructions. They can make the source code shorter.)

To use the subroutine, you need to have the string stored in memory. Each assembler has it's own way of assigning data to memory. Here are examples of how Custom Software does it:

```
EL       DB      $4C      ;stores ASCII "L" to memory location "EL"
JK       DW      $4A4B    ;stores $4A to location "JK" & $4B to "JK"+1
MESSAGE  DM      TIME =   ;stores characters starting at location "MESSAGE"
```

DB means Define Byte. DW means Define Word (two bytes), and DM means Define Message. DB and DW are pretty standard, but on some assemblers DM is STR, for STRing. DB, DW and DM are all assembler directives. They tell the assembler that there is data here, and not an instruction. The assembler stores the appropriate value(s) in memory for you.

Notice the words "EL", "JK" and "MESSAGE" in the first column of each assembler directive. The first column is called the label field, and those words are labels. The use of labels is discussed below.

Now that we have the message stored in memory, we can use the ROM subroutine at $5A58. According to Custom Software's manual, there are two requirements that must be met (entry conditions). The data to be displayed must be terminated with a zero. The subroutine will display character after character until it reaches a zero, and then it will RETurn to your program. Remember, it is looking for the value zero, not the ASCII value for zero, which is $30. We must therefore add:

```
        DB      $00      ;store the value zero as a terminator
```

after the message line above.

The other entry condition is that HL should point to the characters to be displayed. That means the HL register needs to contain the address of the message. What is the address of the message? If it is the first line of the program, it will be wherever the program is ORG'ed. If it isn't, we would then have to count the number of bytes from the beginning of the program, and add the ORG value to that. If you change the program, you would then have to recalculate each address.

Luckily, there is a easier way, using labels. In the example above, MESSAGE is a label. When the source code is assembled, the assembler keeps track of all the labels. In this case, it would know the memory location of the label MESSAGE. You can then use "MESSAGE" whenever you mean "the memory location of the label MESSAGE".

Now we can load the HL register with the address of the message:

```
        LXI     HL,MESSAGE      ;load HL with the address of the message
```

and then call the subroutine to display it:

```
        CALL    $5A58   ;display message
```

We don't even have to know what address MESSAGE stands for, the assembler will take care of it for us. However, most assemblers will give you a list of all labels and their values at the end of the source code listing.

Suppose now we wanted to call the subroutine at location $5A58 "DISPLAY". If we could tell the assembler that the label DISPLAY means $5A58, then we could use the label instead. We can do that using the EQUate directive.

The EQUate directive tells the assembler to assign a value to the label of the EQUate directive. For example:

```
DISPLAY EQU     $5A58   ;assign $5A58 to the label DISPLAY
```

at the beginning of your program will tell the assembler to substitute $5A58 wherever it sees DISPLAY.

Program 2 does the same thing as program 1, but a little more elegantly. Here we use labels, and the ROM subroutine at $5A58.

```
        ;PRTIME2
        ;Nov 12, 1984
;
        ORG     $DAC0
        ENT     $DAC0
;
; These are all ROM subroutines
DISPLAY EQU     $5A58   ;print message pointed to by HL
LCD     EQU     $4B44   ;print character in register A
CHGET   EQU     $12CB   ;wait for keypress
MENU    EQU     $5797   ;main MODEL 100 menu
;
SECS    EQU     $F933   ;memory location for seconds
;
BEGIN   LXI     HL,MESSAGE      ;set HL pointer to start of message data
        CALL    DISPLAY         ;display message
;
        LXI     HL,SECS+1       ;SECS+1 = $F933+1 = $F934
        MOV     A,M
        CALL    LCD
        LXI     HL,SECS
        MOV     A,M
        CALL    LCD
;
        CALL    CHGET
        JMP     MENU
;
MESSAGE DM      TIME =          ;message data
        DB      $00             ;terminator
;
        END
```

PROGRAM 2. Print seconds to screen (using labels).

In program 2 you can see that you can perform arithmetic on labels. Some assemblers can evaluate very complex expressions, while others can handle only addition and subtraction. Check the documentation for your particular assembler.

. .. On page 80 of the Radio Shack Technical Manual is listed the addresses for several useful display subroutines. Also listed are the cursor locations. By moving different values into these two locations, you can start your printing anywhere on the screen. Experimenting is fun, but don't forget to be prepared for those cold starts!

(Thanks to Greg Susong of Custom Software for permission to use information taken from the Custom Software Assembler Manual.)